CIS 630 - Fall 2004
Distributed Systems

Lecture 8
Distributed Transactions

University of Oregon
Department of Computer and Information Science
Business and Logistics

☐ Programming exercise due
   ☑ Some demonstrations done yesterday
   ☑ Need to schedule remainder
   ☑ I will be in lab from 2:30 - 3:00 pm today

☐ Project proposal responses
   ☑ In your email today
   ☑ All project proposals look good

☐ Lecture slides
   ☑ Uploaded sometime today
Acknowledgements

- Some material taken from author’s teaching slides based on Distributed Systems: Concepts and Design book
- Some figures taken from Distributed Systems: Concepts and Design book
Lecture Objectives

- Present the problem of distributed transactions
- Discuss the two phase commit protocol
- Discuss distributed concurrency control
  - Based on local concurrency control at each server
  - Distributed deadlock detection is complex
- Discuss distributed deadlock detection
  - A centralized solution is not scalable
  - Phantom deadlocks are a problem
  - Use edge chasing
- Discuss recovery to ensure atomicity and durability
  - Transaction log and checkpoints
Distributed Transactions

- Have been considering single server
- In general, data items belonging to a service may be distributed among several servers
- Client transactions involve multiple servers
  - Directly by requests made by a client
  - Indirectly via requests made by servers
- *Distributed transaction*
  - Any transaction whose activities involve multiple servers is a transaction
- Client transactions that involve multiple servers indirectly may be modelled as nested transactions
Distributed Transactions (continued)

- Refers to a flat or nested transaction that accesses objects managed by multiple servers
- Transaction commitment is still a problem
- When a distributed transaction comes to an end
  - Either all of the servers commit the transaction
  - Or all of them abort the transaction
- One of the servers is *coordinator*
  - Ensure the same outcome at all of the servers
- The “two-phase commit protocol” is the most commonly used protocol for achieving this
Structuring of Distributed Transactions

- Simple distributed transaction
  - Client makes requests to more than one server
  - Each server carries out the client’s requests without invoking operations on other servers
  - Each transaction accesses servers’ data items sequentially
  - When locking is used, a transaction can only be waiting for one data item at a time

- Nested transaction
  - Server invokes operations on other servers
  - Hierarchy of nested transactions
Distributed Transaction Types

(a) Flat transaction

(b) Nested transactions

Issues?
Case Study: Nested Banking Transaction

$T = \text{openTransaction}$

- $\text{openSubTransaction} a.\text{withdraw}(10)$;
- $\text{openSubTransaction} b.\text{withdraw}(20)$;
- $\text{openSubTransaction} c.\text{deposit}(10)$;
- $\text{openSubTransaction} d.\text{deposit}(20)$;
- $\text{closeTransaction}$

Issues?
Requirements for Distributed Transactions

- Atomicity
  - Either all of the servers involved commit
  - Or all of them abort
  - Coordinator ensures the same outcome
  - Depends on protocol chosen
  - “Two-phase commit protocol” is common

- Concurrency control
  - Local control to ensure transactions are serializable
  - Must be serialized globally as well
  - Extention of concurrency control methods for single server transactions
Coordinator of a Distributed Transaction

- Distributed servers need to coordinate their actions when the transaction commits.
- Client sends *OpenTransaction* request to server.
- Server returns *transaction ID (TID)*
  - Must be unique within a distributed system.
  - Server ID + unique ID within server.
- First server in the transaction becomes *coordinator*.
  - Responsible for committing or aborting.
  - Responsible for adding other servers (*participants*).
    - Cooperates with coordinator in commit protocol.
  - Records list of worker and coordinator ID.
AddServer Transactional Service Function

- AddServer(TID, Server ID of coordinator)
  - Informs server involved in transaction TID
- AddServer must be used by the client before any operations are requested in a server not yet joined
  - Supplies transaction ID
  - Supplies transaction coordinator server ID
- Receipt of AddServer
  - Initializes local transaction
  - Sends NewServer request to coordinator
  - NewServer(Trans, Server ID of worker)
**Distributed Banking Transaction**

$T = \text{openTransaction}$

$a.\text{withdraw}(4);$
$c.\text{deposit}(4);$
$b.\text{withdraw}(3);$  
$d.\text{deposit}(3);$  
$\text{closeTransaction}$

Note: the coordinator is in one of the servers, e.g. BranchX
Coordination and Transaction Completion

- Coordinator and workers knowing each other enables them to collect information needed at commit time
- Distribution of servers in a transaction can be made transparent to user-level programs
  - Record ID of server that opens transaction
  - Issue AddServer when new server joins with TID
- CloseTransaction or AbortTransaction
  - Called when transaction ends
Atomic Commit Protocols

- Transaction ends when client requests that the transaction should be committed or aborted
- Require atomicity

- **One-phase atomic commit** protocol
  - Coordinator communicates the commit or abort request to all the servers in the transaction
  - Continue repeating request until all had acknowledged

- One-phase atomic commit is inadequate
  - Client requests a commit
  - Does not allow server to unilaterally abort
  - Servers must be able to abort in certain situations
Commit Protocols

Commit protocols are designed to work in
- Asynchronous system
  - messages may take a very long time
- Servers may crash
- Messages may be lost
- Assume corrupt and duplicated messages are removed
- No byzantine faults
  - servers either crash or they obey their requests
Two-Phase Commit Protocol

- Designed to allow any server to abort its part of the transaction
- Due to atomicity, if one part of a transaction is aborted, the whole transaction must be aborted
- First phase (commit)
  - Each server votes for transaction to be committed or aborted
  - Once a server votes commit, it cannot abort
  - Server must ensure it can commit before voting
  - Transaction is said to be in a prepared state
Two-Phase Commit Protocol (continued)

- Second phase
  - Every server carries out the joint decision
  - If any one server votes to abort, then the decision must be to abort
  - If all servers vote to commit, then the decision is to commit the transaction

- The problem is to ensure that all the servers vote and that they all reach the same decision
  - Simple with no errors
  - Protocol must work correctly in face of failures, lost messages, temporary loss of communication
More on Two-Phase Commit Protocol

- A client’s request to commit/abort directed to coordinator
- Client abort or server transaction abort
  - Coordinator informs workers immediately
- Two-phase commit protocol comes into play when client asks coordinator to commit
- First phase (commit)
  - Coordinator asks workers if they are prepared
  - Coordinator tells workers to commit (abort)
  - Server-to-server operations
More Two-Phase Commit Protocol (continued)

- Voting phase and completion phase
- Apparently straightforward protocol could fail due to one or more of the servers failing or due to a breakdown in communication
- Each server saves information relating to the two-phase commit protocol in permanent storage
- Timeout actions are included in the protocol
  - Various stages at which a server cannot progress its part of the protocol until it receives another request or reply from one of the other servers
- 2PC is an example of a consensus protocol
Operations for Two-Phase Commit Protocol

`canCommit?(trans) -> Yes / No`
Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

`doCommit(trans)`
Call from coordinator to participant to tell participant to commit its part of a transaction.

`doAbort(trans)`
Call from coordinator to participant to tell participant to abort its part of a transaction.

`haveCommitted(trans, participant)`
Call from participant to coordinator to confirm that it has committed the transaction.

`getDecision(trans) -> Yes / No`
Call from participant to coordinator to ask for the decision on a transaction after it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages.
Two-Phase Commit Protocol

Phase 1 (voting phase):
1. The coordinator sends a canCommit? request to each of the participants in the transaction.
2. When a participant receives a canCommit? request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

Phase 2 (completion according to outcome of vote):
3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are Yes the coordinator decides to commit the transaction and sends a doCommit request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends doAbort requests to all participants that voted Yes.
4. Participants that voted Yes are waiting for a doCommit or doAbort request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a haveCommitted call as confirmation to the coordinator.
Communication in Two-Phase Commit Protocol

Coordinator

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>prepared to commit (waiting for votes)</td>
</tr>
<tr>
<td>3</td>
<td>committed</td>
</tr>
<tr>
<td></td>
<td>done</td>
</tr>
</tbody>
</table>

Participant

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>prepared to commit (uncertain)</td>
</tr>
<tr>
<td>4</td>
<td>committed</td>
</tr>
</tbody>
</table>

canCommit?

Yes

doCommit

haveCommitted
Timeouts

- Worker votes *Yes* and waits for coordinator to report on the outcome.
- Worker is uncertain of the outcome and cannot proceed.
- Worker makes *GetDecision* request:
  - Get reply to continue protocol
  - Wait for reply
- Worker could obtain decision cooperatively:
  - Useful when coordinator has failed
  - Still need to get out of uncertain states
Timeouts (continued)

- Worker can be delayed when carried out all client requests, but not yet received \textit{CanCommit}? from coordinator
  - Worker can decide to \textit{Abort} unilaterally
- Coordinator may be delayed waiting for votes from the workers
  - May decide to abort the transaction
  - Announce \textit{AbortTransaction} to the workers who have already sent their votes
  - Tardy workers voting \textit{Yes} will be ignored
Performance of Two-Phase Commit Protocol

☐ All goes well ($N$ servers)
  ☐ ($N-1$) $CanCommit$? messages and replies
  ☐ ($N-1$) $DoCommit$ messages
  ☐ Message cost: proportional to $3N$
  ☐ Time cost: three rounds of messages
  ☐ $HaveCommitted$ not counted

☐ Worst case
  ☐ Arbitrarily many server and communication failures
  ☐ Can tolerate succession of failures
  ☐ Guarantees to complete eventually
Performance (continued)

- Considerable delay to workers in uncertain states
- Occurs when the coordinator has failed and cannot reply to *GetDecision* requests from workers
- Three-phase commit protocols have been designed to alleviate delays
Summary of Two-Phase Commit Protocol

- Distributed transactions involve several servers
- Atomicity requires that the servers participating in a distributed transaction either all commit or all abort
- Atomic commit protocols are designed to achieve this effect, even if servers crash during their execution
- 2PC protocol allows a server to abort unilaterally
  - Includes timeout actions to deal with delays due to servers crashing
  - 2PC protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually
Distributed Concurrency Control

- Collection of servers of distributed transactions
  - Jointly responsible for ensuring transaction performed in serial equivalent manner
  - Each manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions

- Transaction $T$ occurring before transaction $U$ at one server, it must be in that order at all servers

- Mechanisms
  - Locking
  - Timestamp ordering
  - Optimistic concurrency control
Locks

☐ In a distributed transaction, the locks on an object are held by the server that manages it
  ✗ The local lock manager decides whether to grant a lock or make the requesting transaction wait
  ✗ It cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction
  ✗ The objects remain locked and are unavailable for other transactions during the atomic commit protocol
    ➢ an aborted transaction releases its locks after phase 1 of the protocol.
### Interleavings of Transactions U, V, and W

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>V</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.deposit(10)</td>
<td>lock D</td>
<td>b.deposit(10)</td>
<td>lock B at Y</td>
</tr>
<tr>
<td>a.deposit(20)</td>
<td>lock A at X</td>
<td>c.deposit(30)</td>
<td>lock C at Z</td>
</tr>
<tr>
<td>b.withdraw(30)</td>
<td>wait at Y</td>
<td>c.withdraw(20)</td>
<td>wait at Z</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a.withdraw(20)</td>
</tr>
</tbody>
</table>
Distributed Deadlocks

- Single server transactions can experience deadlocks
  - Prevent or detect and resolve
  - Use of timeouts is clumsy
  - Detection is preferable
    - it uses *wait-for* graphs

- Distributed transactions lead to distributed deadlocks
  - In theory can construct global *wait-for* graph from local ones
  - A cycle in a global *wait-for* graph that is not in local ones is a distributed deadlock
Distributed Deadlock Detection

- A global wait-for graph can in theory be constructed from local ones
- There can be a cycle in the global wait-for graph that is not in any single local one
  - Distributed deadlock
  - Deadlock iff there is a cycle in the wait-for graph
- Detection of distributed deadlock requires a cycle to be found in global transaction wait-for graph distributed among the servers
  - Local wait-for graphs
  - Communication required between servers
Distributed Deadlock
Distributed Deadlock Solutions

- Centralize deadlock detection
  - One server is global deadlock detector
  - Collects local wait-for graphs
  - Builds global wait-for graph and finds cycles
  - Decides how to resolve deadlock
  - Inform servers as to the transactions to be aborted

- Issues
  - Centralized approach has poor reliability
  - Transmitting local wait-for graphs is high
Phantom Deadlocks

- Deadlock detected but not really a deadlock
- Information about wait-for relationships between transactions eventually collected in one place
- Chance that transaction holding a lock will release it and no deadlock will exist
- Simple phantom deadlocks will not arise if two-phase locks are used
- A phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure
Local and Global Wait-For Graphs

local wait-for graph

\[ T \rightarrow U \]

\[ X \]

local wait-for graph

\[ V \rightarrow T \]

\[ Y \]

global deadlock detector

\[ T \rightarrow U \rightarrow V \]
Edge Chasing (Path Pushing)

- Global wait-for graph not constructed
  - Servers involved each know some edges
- Servers attempt to find cycles by forwarding messages called *probes*
  - Follow edges of the graph throughout system
  - Contains transaction wait-for relationships representing a path in the global wait-for graph
- When should a server send out a probe?
- At any point, a transaction can be either active or waiting at just one of these servers
Edge Chasing (Path Pushing) (continued)

- Coordinator records active or waiting for a data item and workers can get this information
  - Lock managers inform coordinators when transactions start waiting or become active

- Coordinator informs workers when transaction is aborted and locks can be released and edges removed in local wait-for graphs

- Edge chasing has three steps:
  - *Initiation*: sending out probes on waiting events
  - *Detection*: receiving probes and detecting cycles
  - *Resolution*: aborting transactions to break deadlock
Edge Chasing (Path Pushing) (continued)

- **Initiation**
  - $T$ waits for $U$ where $U$ is waiting to access a data item at another server
  - Send probe containing edge $<T\rightarrow U>$ to server where $U$ is blocked
  - If $U$ sharing a lock, probes sent to holders of lock

- **Detection**
  - Receive $<T\rightarrow U>$ : check to see if $U$ also waiting
  - If so, transaction it waits for is added to the probe $<T\rightarrow U\rightarrow V>$ and probe is forward if necessary

- **Resolution by breaking the cycle**
Edge Chasing (Path Pushing) (continued)

- Before a server transmits a probe to another server, it consults the coordinator of the last transaction in the path to find out whether the latter is waiting for another data item elsewhere.

- Most often servers send probes to transaction coordinators which then forward them to the server of the data item the transaction is waiting for.

- Deadlocks should be found provided waiting transactions do not abort and there are no failures:
  - $2(N-1)$ messages sent for a cycle involving $N$ transactions.
Probes Transmitted to Detect Deadlocks

$W \rightarrow U \rightarrow V \rightarrow W$

Deadlock detected

$Z$

Waits for

$V \rightarrow U \rightarrow V$

Held by

$B$

Initiation

$W \rightarrow U \rightarrow V$

Waits for

$U \rightarrow X$

Held by
Transaction Priorities

- Every transaction involved in a deadlock cycle can cause deadlock detection to be initiated.
- Detection may happen at several different servers.
- More than one transaction can be aborted.
- Transactions are given priorities such that all transactions are totally ordered:
  - Ensures that only one transaction aborted in a cycle.
  - Transaction with the lowest priority is aborted.
  - Might be used to reduce the number of situations that cause deadlock detected to be initiated.
Two Probes Initiated

(a) initial situation

(b) detection initiated at object requested by $T$

(c) detection initiated at object requested by $W$

Waits for $V$
Waits for $U$
Waits for $W$
Transaction Priorities (continued)

- Detection initiated only when a higher priority transaction starts to wait for a lower priority one
  - Reduce # of probe messages by half
- Priorities used to reduce # of probes forwarded
  - Probes should travel “downhill” from transactions with higher priorities to transactions lower
  - Servers do not forward any probe to a holder with higher priority than the initiator
  - The order in which transactions start waiting can determine whether or not deadlock will be detected
Transaction Priorities (continued)

- **Probe queue**
  - Save copies of all the probes received on behalf of each transaction

- **Transaction starts waiting for a data item**
  - Forwards the probes in its queue to the server of the data item which propagates probes downhill

- **Algorithm requiring probes stored in probe queues**
  - Arrangements to pass on probes to new holds
  - Discard probes for committed/aborted transactions
  - Adds must to the complexity of edge-chasing
Probes Travel Downhill

(a) V stores probe when U starts waiting

(b) Probe is forwarded when V starts waiting